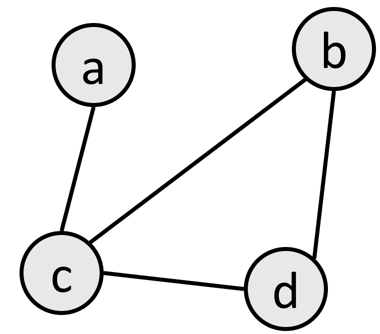
CS60020: Foundations of Algorithm Design and Machine Learning

Sourangshu Bhattacharya

Graphs

- **graph**: A data structure containing:
  - a set of **vertices** V, *(sometimes called nodes)*
  - a set of **edges** E, where an edge represents a connection between 2 vertices.
    - Graph $G = (V, E)$
    - an edge is a pair (v, w) where v, w are in V

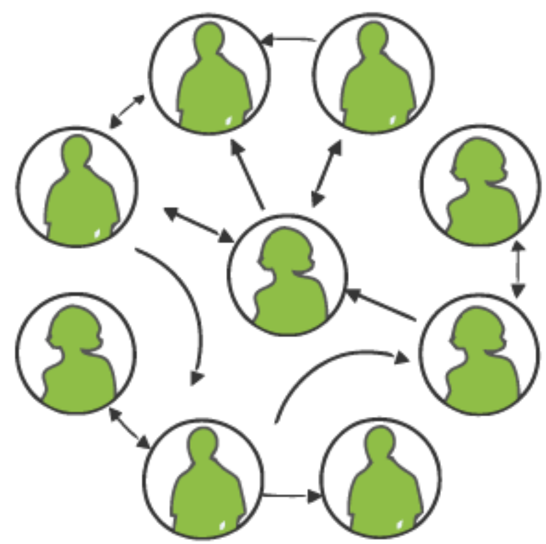


- the graph at right:
  - $V = \{a, b, c, d\}$
  - $E = \{(a, c), (b, c), (b, d), (c, d)\}$

- **degree**: number of edges touching a given vertex.
  - at right: a=1, b=2, c=3, d=2

Graph examples

- For each, what are the vertices and what are the edges?
  - Web pages with links
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Facebook friends
  - Course pre-requisites
  - Family trees
  - Paths through a maze

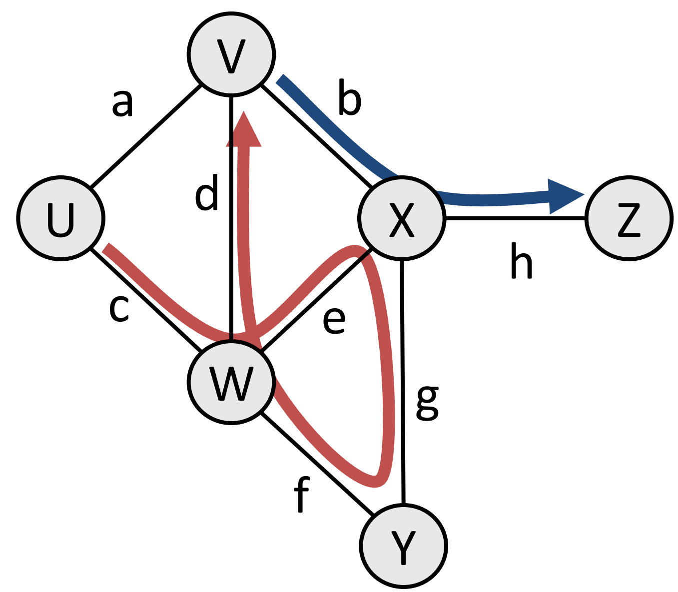


Paths

- **path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  – can be represented as vertices visited, or edges taken
  – example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  – What are two paths from U to Y?

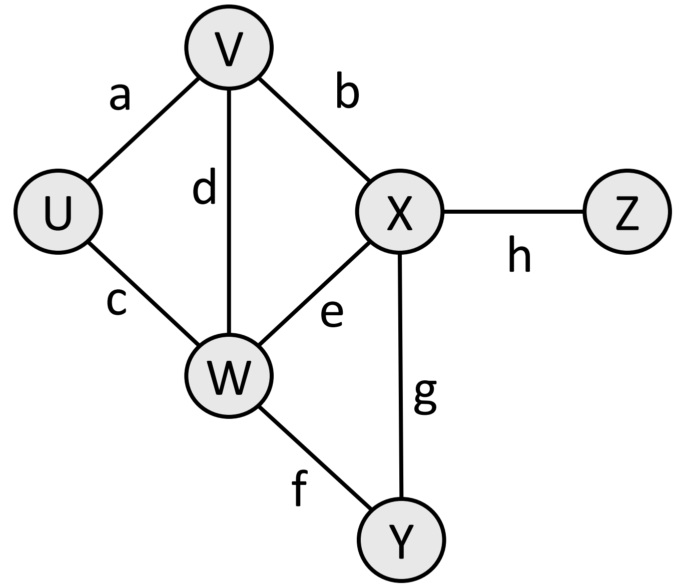
- **path length**: Number of vertices or edges contained in the path.

- **neighbor** or **adjacent***:* Two vertices connected directly by an edge.
  – example: V and X



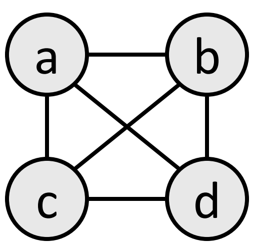
Reachability, connectedness

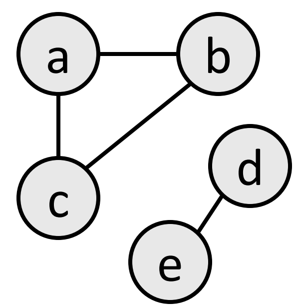
- **reachable**: Vertex *U* is *reachable* from *V* if a path exists from *U* to *V*.



- **connected**: A graph is *connected* if every vertex is reachable from any other.
  - Is the graph at top right connected?

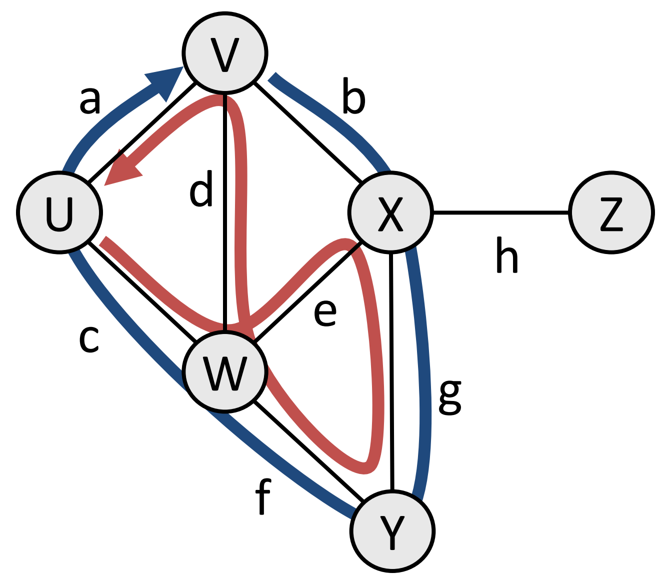
- **strongly connected**: When every vertex has an edge to every other vertex.





Loops and cycles

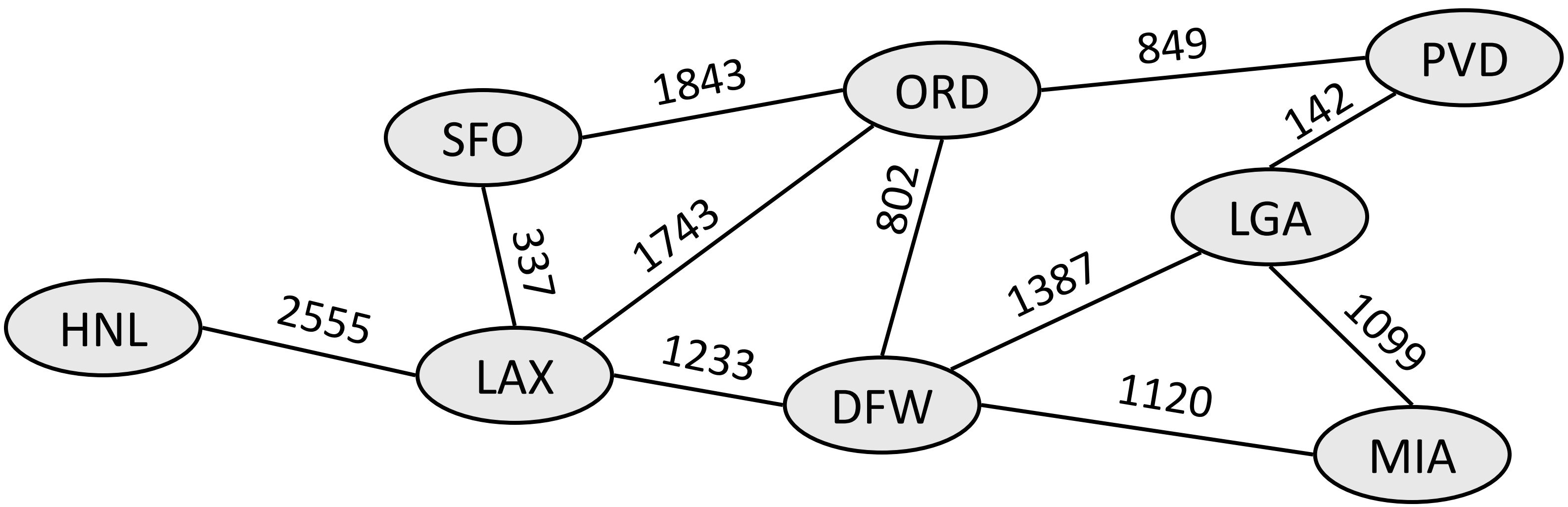
- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.
  - **acyclic graph**: One that does not contain any cycles.
- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.



Weighted graphs

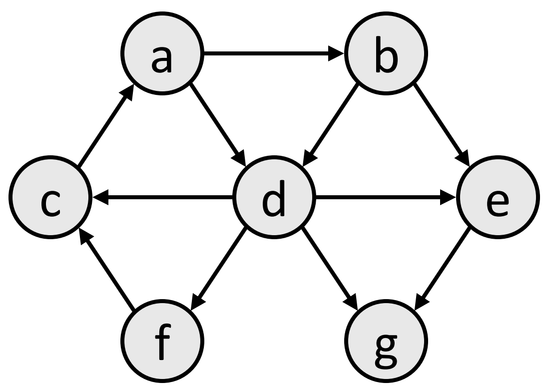
- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.

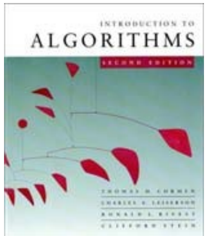
- *example*: graph of airline flights, weighted by miles between cities:



Directed graphs

- **directed graph** ("digraph"): One where edges are *one-way* connections between vertices.
  - If graph is directed, a vertex has a separate in/out degree.
  - A digraph can be weighted or unweighted.
  - Is the graph below connected? Why or why not?





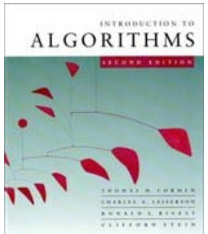
Graphs (review)

Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

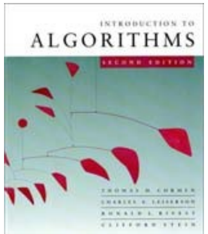
In either case, we have $|E| = O(V^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$, which implies that $\lg |E| = \Theta(\lg V)$.



Adjacency-matrix representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

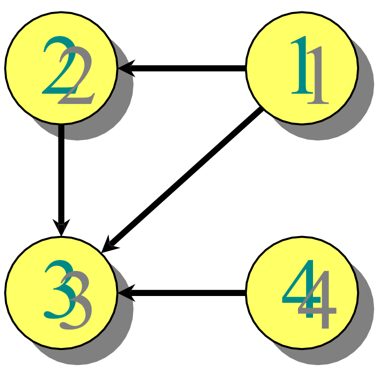
$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in \mathrm{E}, \\ 0 & \text{if } (i, j) \notin \mathrm{E}. \end{cases}$$



Adjacency-matrix representation

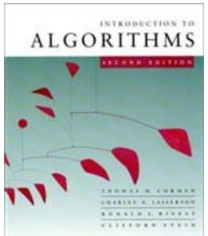
The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,.\,.\, n, 1 \,.\,.\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$



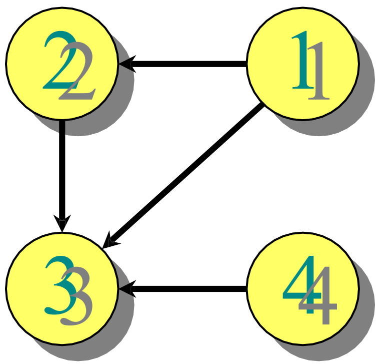
A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(V^2)$ storage $\Rightarrow$ ***dense*** representation.

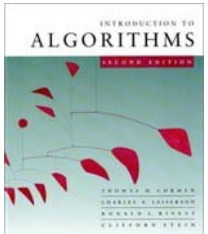


Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.

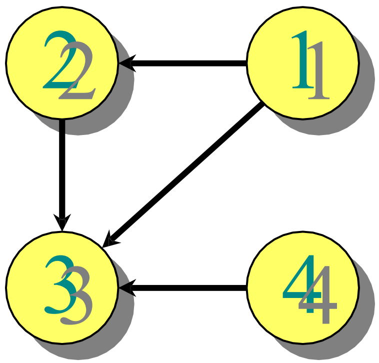


$$Adj[1] = \{2, 3\}$$
$$Adj[2] = \{3\}$$
$$Adj[3] = \{\}$$
$$Adj[4] = \{3\}$$



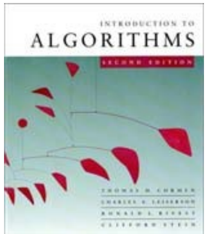
Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



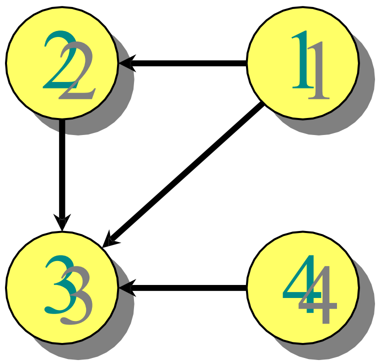
$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.



Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.
For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

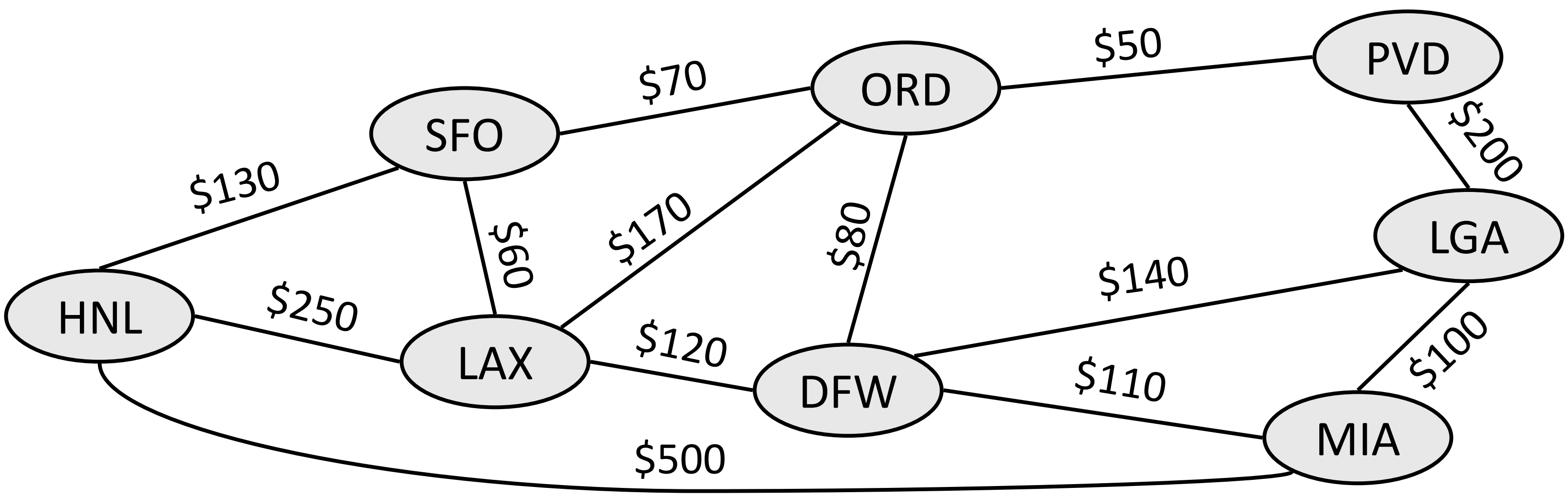
Handshaking Lemma: $\sum_{v \in V} \mathrm{Adj}[v] = 2\,|\mathrm{E}|$ for undirected graphs $\Rightarrow$ adjacency lists use $\Theta(V + E)$ storage — a ***sparse*** representation (for either type of graph).

GRAPH SEARCH

Searching for paths

- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).

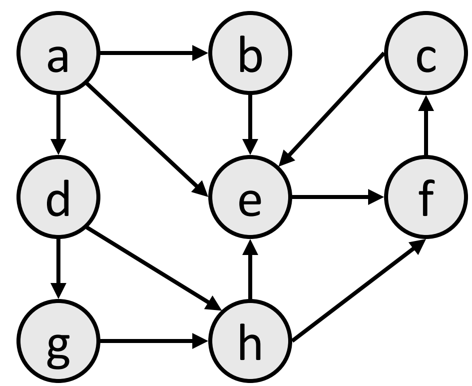
- What is the shortest path from MIA to SFO?
  Which path has the minimum cost?



Depth-first search

- **depth-first search** (DFS): Finds a path between two vertices by exploring each possible path as far as possible before backtracking.
  - Often implemented recursively.
  - Many graph algorithms involve *visiting* or *marking* vertices.

- Depth-first paths from *a* to all vertices (assuming ABC edge order):
  - to b: {a, b}
  - to c: {a, b, e, f, c}
  - to d: {a, d}
  - to e: {a, b, e}
  - to f: {a, b, e, f}
  - to g: {a, d, g}
  - to h: {a, d, g, h}



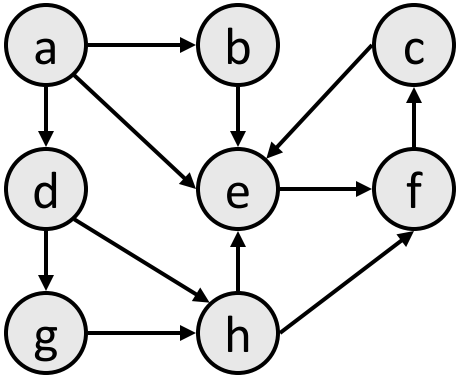
DFS pseudocode

function **dfs**(v_1, v_2):
  dfs(v_1, v_2, { }).

function **dfs**(v_1, v_2, *path*):
  path += v_1.
  mark v_1 as visited.
  if v_1 is v_2:
    a path is found!

  for each unvisited neighbor n of v_1:
    if dfs(n, v_2, *path*) finds a path: a path is found!

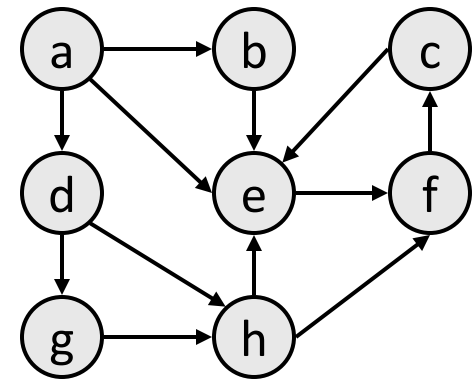
  path -= v_1. // path is not found.



- The *path* param above is used if you want to have the path available as a list once you are done.
  - Trace dfs(*a*, *f*) in the above graph.

DFS observations

- *discovery*: DFS is guaranteed to find *a* path if one exists.



- *retrieval*: It is easy to retrieve exactly what the path is (the sequence of edges taken) if we find it

- *optimality*: not optimal. DFS is guaranteed to find a path, not necessarily the best/shortest path
  - Example: dfs(a, f) returns {a, d, c, f} rather than {a, d, f}.